On the Semantics of ReFLect as a Basis for a Reflective Theorem Prover

Tom Melham[1], Raphael Cohn[2], and Ian Childs[1]

[1] University of Oxford
[2] The Rockefeller University

Abstract. This paper explores the semantics of a combinatory fragment of *reFLect*, the λ-calculus underlying a functional language used by Intel Corporation for hardware design and verification. *ReFLect* is similar to ML, but has a primitive data type whose elements are the abstract syntax trees of *reFLect* expressions themselves. Following the LCF paradigm, this is intended to serve as the object language of a higher-order logic theorem prover for specification and reasoning—but one in which object- and meta-languages are unified. The aim is to intermix program evaluation and logical deduction through reflection mechanisms. We identify some difficulties with the semantics of *reFLect* as currently defined, and propose a minimal modification of the type system that avoids these problems.

1 Introduction

ReFLect is a strongly typed, functional programming language, similar to ML, with certain reflection features for applications in hardware design and verification [4]. The language was designed by researchers at Intel Corporation and forms the basis for Intel's Forte formal verification environment [9]. Forte has been used at Intel to attack many challenging verification problems for real-world processor designs; one impressive achievement is the verification, without simulation, of much of the Core i7 processor execution cluster [6].

Reflection is supported through a primitive data type, *term*, whose values are *reFLect* abstract syntax trees. A *quotation* $\langle e \rangle$ has type *term* and denotes the syntax tree of the term e. In Forte, this type is the basis for a higher-order logic theorem prover that is similar to HOL [3]. In systems like HOL, the higher-order logic 'object language' is built on top of the λ-calculus, following Church [1]. The syntax of the logic is represented by an algebraic data-type in a functional programming 'meta-language'. *ReFLect* is designed to unify the object and meta languages in this kind of enterprise. The aim is for the *same* λ-calculus to be the core of both logic and meta-language. Moreover, it should be possible to move freely between *evaluation* in the interpreter and *deduction* in the theorem prover. For example, the reduction rules of *reFLect* align with inference rules of its logic, so proof can be done just by evaluation: to prove a theorem $\langle P \rangle$, simply strip off the quotes and check that the interpreter evaluates P to 'true'.

The logical soundness of this depends on having the right formal semantics to justify these reflection rules. There exists an operational (i.e. reduction)

semantics for full *reFLect*, including quotation evaluation [4], but for logic a denotational semantics is needed. Krstić and Matthews have published one [7], but this omits the crucial reflection constructs that bridge logic and evaulation.

In this paper, we analyse the semantics of a simplified system, *Combinatory ReFLect*, that has some core features of full *reFLect* and includes the reflection constructs missing from the semantics in [7]. This gives a simple setting to investigate the problem while side-stepping the technicalities of variable binding in the full language. We find that even our variable-free language does not support the semantics we need, and that any reasonable logic built on *reFLect* will be inconsistent. Our proposed solution is a modification of the language's type system that stratifies the semantics enough to avoid the problem.

2 Combinators with Reflection

Combinatory ReFLect (CR) is a variable-free system of combinatory logic that includes essentially the same reflection constructs, quotations, and typing system as full *reFLect*. Indeed, CR can be viewed as a sublanguage of full *reFLect*. The syntax of CR includes explicitly-typed versions of the combinators I, K and S, along with reflective operators value, app and lift. CR also supports quotation of any term e in the language, written $\langle e \rangle$.

The types of CR terms, *Ty*, are defined inductively by

$$\sigma \;:=\; \mathit{bool} \;\mid\; \mathit{unit} \;\mid\; \mathit{term} \;\mid\; \sigma_1 \to \sigma_2$$

And the terms of CR, *Exp*, are given by

$$e \;:=\; \mathsf{I}_\sigma \;\mid\; \mathsf{K}_{\sigma,\tau} \;\mid\; \mathsf{S}_{\sigma,\tau,\upsilon} \;\mid\; \mathsf{value}_\sigma \;\mid\; \mathsf{lift} \;\mid\; \mathsf{app} \;\mid\; e_1\, e_2 \;\mid\; \langle e \rangle \qquad (1)$$

where σ, τ and υ range over types. We refer to terms of the form '$e_1\ e_2$' as *applications*, and terms of the form '$\langle e \rangle$' as *quotations*.

A quotation is an object-language phrase of CR that, semantically, denotes the abstract syntax tree of the term inside the quotes. A quotation $\langle e \rangle$ is semantically different from the term e. Unquoted terms that evaluate to the same value, for example $(\mathsf{S}\,\mathsf{K})\,\mathsf{K}\ e$ and $\mathsf{I}\ e$, are semantically equal. But $\langle (\mathsf{S}\,\mathsf{K})\,\mathsf{K}\ e \rangle$ and $\langle \mathsf{I}\ e \rangle$ are semantically distinct; they denote different syntax trees.

To explain the intended semantics of value_σ, lift, and app, we suppose, just for a moment, that normal forms exist in some reduction system for CR. Of course we have not *established* this yet. The intended semantics, then, is that $\mathsf{value}_\sigma\ \langle e \rangle$ should denote the normal form or evaluation of e of type σ. This is obviously problematic if e contains free variables, so the operational semantics of *reFLect* allows reduction of $\mathsf{value}_\sigma\ \langle e \rangle$ only for closed e [4, p. 187]. The lift function reifies values into representative syntax and, in full *reFLect*, applies only to closed terms that possess a canonical 'name' for their value [4, §8.2]. For example, both $\mathsf{lift}\ 2$ and $\mathsf{lift}\ (1+1)$ reduce to $\langle 2 \rangle$. In CR, we adopt an even more conservative semantics for lift: it applies only to terms of type *term*, and we can reduce only terms of the form $\mathsf{lift}\ \langle e \rangle$. It seems obviously harmless to suppose we might have

$$\frac{}{\mathsf{I}_\sigma : \sigma \to \sigma} \qquad \frac{}{\mathsf{K}_{\sigma,\tau} : \sigma \to \tau \to \sigma} \qquad \frac{}{\mathsf{S}_{\sigma,\tau,\upsilon} : (\sigma \to \tau \to \upsilon) \to (\sigma \to \tau) \to \sigma \to \upsilon}$$

$$\frac{}{\mathsf{value}_\sigma : \mathit{term} \to \sigma} \qquad \frac{}{\mathsf{lift} : \mathit{term} \to \mathit{term}} \qquad \frac{}{\mathsf{app} : \mathit{term} \to \mathit{term} \to \mathit{term}}$$

$$\frac{e_1 : \sigma \to \tau \quad e_2 : \sigma}{e_1\ e_2 : \tau} \qquad \frac{e : \sigma}{\langle e \rangle : \mathit{term}}$$

Fig. 1. Typing rules for CR.

$$\mathsf{I}_\sigma\ e \Rightarrow e \qquad \mathsf{K}_{\sigma,\tau}\ e_1\ e_2 \Rightarrow e_1 \qquad \mathsf{S}_{\sigma,\tau,\upsilon}\ e_1\ e_2\ e_3 \Rightarrow e_1\ e_3\ (e_2\ e_3)$$

$$\text{if } e : \sigma, \text{ then } \mathsf{value}_\sigma\ \langle e \rangle \Rightarrow e \qquad \mathsf{lift}\ \langle s \rangle \Rightarrow \langle \langle s \rangle \rangle$$

$$\text{if } e_1\ e_2 \text{ is well typed, then } \mathsf{app}\ \langle e_1 \rangle\ \langle e_2 \rangle \Rightarrow \langle e_1\ e_2 \rangle$$

Fig. 2. Some Reductions in CR.

a function that takes $\langle e \rangle$ to $\langle \langle e \rangle \rangle$. Finally, in what follows we will need to apply one *term* to another; this is native in full *reFLect*, but in CR we have an ad-hoc combinator, app.

Figure 1 shows the type system of CR. The judgement $e : \sigma$ means that the term e has type σ. Note that the type system does not rule out the formation of a term '$\mathsf{value}_\sigma\ \langle e \rangle$' where e does not have type σ. A check for this kind of type mismatch is done only at runtime in CR, during reduction. In Figure 2 we show part of the reduction system for CR. (We omit only the rules that close $\Rightarrow$ under reflexivity, transitivity, and congruence.)

2.1 Denotational Semantics

The reflective operators value_σ and lift are key components of the *reFLect* language. They serve as the essential bridge betwen deductive logic and program evaluation in Forte. It is these operators, omitted from the semantics in [7], that make the denotational semantics of *reFLect* difficult, as we now show.

Suppose we have a semantic function, $[\![-]\!] \in \mathit{Exp} \to U$ that maps each term to an element of some universe U. A quotation $\langle e \rangle$ represents the abstract syntax tree of its contents e. So the formal denotation of a quoted term $[\![\langle e \rangle]\!]$, where e is a closed term, can just be the *reFLect* term e itself, an element of $\mathit{Exp} \subseteq U$. And, indeed, this is how it is defined in [7]. With the usual semantics of function application, we would therefore expect for closed $e : \sigma$ that

$$[\![\mathsf{value}_\sigma\ \langle e \rangle]\!] = [\![\mathsf{value}_\sigma]\!]\,[\![\langle e \rangle]\!] = [\![\mathsf{value}_\sigma]\!]\ e = [\![e]\!]$$

So the action of the value_σ function seems to coincide with part of $[\![-]\!]$ itself.

It has been shown that core *reFLect*, without the reflective functions, is confluent and normalizing [7]. The authors of [7] used this normalizing result as a pre-requisite for proving the soundness of their denotational semantics. We now show that adding the reflective functions lift and value_σ destroys this property.

Theorem 1. *Combinatory reFLect is not strongly normalizing.*

Proof. We proceed by constructing a term that has an infinite reduction sequence. We first define a CR expression f such that for all e of type $term \to term$, $f\ \langle e\rangle \Rightarrow \langle e\langle e\rangle\rangle$. In full *reFLect*, f would be $\lambda x.\, \mathsf{app}\ x\ (\mathsf{lift}\ x)$, which in combinators is $\mathsf{S}_{term,term,term}\ \mathsf{app}\ \mathsf{lift}$. It is easy to check that this term is typeable with type $term \to term$ by the rules in Figure 1. We also have the reduction sequence

$$f\ \langle e\rangle = \mathsf{S}\ \mathsf{app}\ \mathsf{lift}\ \langle e\rangle \Rightarrow \mathsf{app}\ \langle e\rangle\ (\mathsf{lift}\ \langle e\rangle) \Rightarrow \mathsf{app}\ \langle e\rangle\ \langle\langle e\rangle\rangle \Rightarrow \langle e\ \langle e\rangle\rangle$$

for any e of type $term \to term$. (We omit type subscripts for readability.)

Now, let the term g of type $term \to term$ be $\mathsf{S}\ (\mathsf{K}\ \mathsf{value}_{term})\ (\mathsf{S}\ (\mathsf{K}\ f)\ \mathsf{I})$; for readability we omit type subscripts. In full *reFLect*, g would be $\lambda x.\, \mathsf{value}_{term}\ (f\ x)$. We immediately see that $g\langle g\rangle$ is well typed. But now we have a circular reduction sequence: $g\ \langle g\rangle \Rightarrow \mathsf{value}_{term}\ (f\ \langle g\rangle) \Rightarrow \mathsf{value}_{term}\ (\langle g\ \langle g\rangle\rangle) \Rightarrow g\ \langle g\rangle$.

2.2 Indefinability in CR

We now show that any higher order logic built on *reFLect* and containing the sublanguage CR will be inconsistent. We will proceed by supposing we do have such a logic, formulated in the usual way deriving from Church [1]. That is, we suppose a logic has been defined on top of CR in the same way that the HOL logic is defined on top of the simply-typed λ-calculus [3]. We then demonstrate inconsistency by a construction inspired by Tarski's Indefinability Theorem [10].

Theorem 2. *In a conventional higher-order logic built on CR, for any term $\Psi : term \to bool$, there is a typeable expression, Γ, such that $\vdash \Gamma = \Psi\langle\Gamma\rangle$.*

Proof. Suppose $\Psi : term \to bool$. Define β of type $term \to bool$ such that for any e of type $term$, $\beta\ e \Rightarrow \Psi(f\ e)$, where $f = \mathsf{S}_{term,term,term}\ \mathsf{app}\ \mathsf{lift}$, as in the proof of Theorem 1. In full *reFLect*, β would just be $\lambda x.\, \Psi(f\ x)$, where x is chosen not to occur free in Ψ. We now let Γ be $\beta\ \langle\beta\rangle$ and prove $\vdash \Gamma = \Psi\ \langle\Gamma\rangle$ in our assumed higher order logic:

$$\begin{array}{ll} \Gamma = \beta\ \langle\beta\rangle & -\text{definition of } \Gamma \\ \quad = \Psi(f\ \langle\beta\rangle) & -\text{reduction of } \beta\ \langle\beta\rangle \\ \quad = \Psi\ \langle\beta\ \langle\beta\rangle\rangle & -\text{reduction of } f\ \langle\beta\rangle \\ \quad = \Psi\ \langle\Gamma\rangle & -\text{definition of } \Gamma \end{array}$$

Corollary 1. *Any conventional higher order logic built on CR is inconsistent.*

Proof. The proof is straightforward and we sketch it here. As discussed, CR contains a truth predicate, value_{bool}. We then have a falsity predicate $\mathsf{isfalse}$ defined to be $\mathsf{S}\ (\mathsf{K}\ \neg)\ (\mathsf{S}\ (\mathsf{K}\ \mathsf{value}_{bool})\ \mathsf{I})$. Taking Ψ in Theorem 2 to be $\mathsf{isfalse}$, we conclude that any logic built on CR can prove $\vdash \Gamma = \mathsf{isfalse}\ \langle\Gamma\rangle$. But then $\mathsf{value}_{bool}\ \langle\Gamma\rangle = \Gamma = \mathsf{isfalse}\ \langle\Gamma\rangle = \neg\ (\mathsf{value}_{bool}\ \langle\Gamma\rangle)$, a contradiction.

3 Stratified Typing for Terms

Analysis of the above results for CR reveals that its fundamental flaw is that all quotations have a single type, *term*. This allows the circularities that prevent normalization and make it logically inconsistent. We now sketch a variant, *Stratified Combinatory ReFLect* (SCR), that avoids these pitfalls. The basic idea is simple: the type of a quotation will carry with it the type of the term inside.

The syntax and reduction semantics of SCR are essentially the same as in CR. The difference is in the types. The types of SCR terms are defined by

$$\sigma \;:=\; \mathit{unit} \;\mid\; (\sigma)\mathit{term} \;\mid\; \sigma_1 \to \sigma_2$$

and the typing rules for the reflective combinators and quotations are

$$\frac{}{\mathsf{value}_\sigma : (\sigma)\mathit{term} \to \sigma} \qquad \frac{}{\mathsf{lift}_\sigma : (\sigma)\mathit{term} \to ((\sigma)\mathit{term})\mathit{term}}$$

$$\frac{}{\mathsf{app} : (\sigma \to \tau)\mathit{term} \to (\sigma)\mathit{term} \to (\tau)\mathit{term}} \qquad \frac{e : \sigma}{\langle e \rangle : (\sigma)\mathit{term}}$$

The other combinators and applications are typed as they are in CR.

This new typing schemes rules out applying a function to its quoted self. It bans the self-application used to define the nonterminating '$g\,\langle g \rangle$' in Theorem 1, and it rules out the '$\beta\,\langle \beta \rangle$' construction used in the proof of Theorem 2.

4 Conclusions and Discussion

This paper has explored the semantics of a reflective combinatory logic that shares key features with *reFLect*, a functional language intended to unify computation and deduction in a practical engineering setting. Our analysis suggests that the type of quotations in *reFLect* must be parameterized by the type of their contents. We speculate that, with this adjustment to the type system, a set-theoretic semantics of full *reFLect*, including reflection, will be possible.

To make *ReFLect* attractive to verification engineers, it has a simple Hindley-Milner type system. This means quotations cannot have types of the form $(\sigma)\mathit{term}$ without making the definitions of certain common functions over terms untypeable. For example, we cannot define $\mathsf{operand}\ \langle e_1\ e_2 \rangle = \langle e_1 \rangle$, since this function would have to have an existential type $(\alpha)\mathit{term} \to \exists\beta.\,(\beta \to \alpha)\mathit{term}$. Also problematic are functions defined recursively over the syntax of terms, which are ubiquitous in theorem-prover code.

Some developments in functional programming, subsequent to the design of *reFLect*, offer a way forward. *Generalized algebraic data types* (GADTs) are a generalization of standard algebraic data types that take a modest step towards dependent types [8]. GADTs allow for algebraic data type constructors to have parameters that can be instantiated to specific types within the body of a function defined over values of the type. We speculate that one could treat quotations and the type $(\sigma)\mathit{term}$ as a GADT in *reFLect*, with the aim of making it possible to define the term-traversing functions needed to implement a theorem prover.

ReFLect currently includes neither GADTs nor $(\sigma)term$. A major project for the future is to develop a full reflective language with GADTs and a theorem prover based on the semantic insights in this paper. A clear first step is to investigate whether GADTs, as found for example in Haskell, are indeed sufficient develop a theorem prover with a parameterized type $(\sigma)term$ of terms. We have done an inital investigation of this, and the answer seems to be 'partly'; the main difficulty seems to be finding a satisfactory treatment of polymorphism.

We are of course aware that there are functional languges and logical calculi with more flexible type systems, among them System F, Coq, and HOL-Omega. But in this industrially-motivated work we have been exploring options that give practicing engineers as simple and intuitive a functional programming language as possible, and so aim to remain close to the Hindley-Milner type system.

More generally, fast object-language evaluation has been a goal of theorem prover designers since the earliest days. Approaches include term data structures that optimise symbolic evaluation by proof, and extraction of programs in standard functional languages [2]. The work presented here is distinguished in attempting direct unification of object- and meta-languages, of symbolic reasoning and direct program execution.

Acknowledgment This work leans heavily on a preliminary sketch of the semantics of full *reFLect* devised in collaboration with Jim Grundy and Sava Krstić [5]

References

1. A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.
2. Coq Development Team. *The Coq Proof Assistant: Reference Manual, V8.4*. Inria, Aug. 2012.
3. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. CUP, 1993.
4. J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *JFP*, 16(2):157–196, Mar. 2006.
5. J. Grundy, T. F. Melham, and S. Krstic. Towards a denotational semantics of *reFLect*. Unpublished presentation slides, 2007.
6. R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In *Computer Aided Verification, CAV 2009*, pages 414–429. Springer, 2009.
7. S. Krstić and J. Matthews. Semantics of the *reFLect* language. In E. Moggi and D. S. Warren, editors, *PPDP*, pages 32–42. ACM, 2004.
8. S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, pages 50–61. ACM, 2006.
9. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE TCAD*, 24(9):1381–1405, Sept. 2005.
10. A. Tarski. The concept of truth in formalized languages. In J. Corcoran, editor, *Logic, Semantics, Metamathematics*, pages 152–278. Hackett Publishing Co., 1983.